Understanding and Enhancing Attribute Prioritization in Fixing Web UI Tests with LLMs

Zhuolin Xu
Concordia University, Canada
zhuolin.xu@mail.concordia.ca

Qiushi Li
Concordia University, Canada
qiushi.li@mail.concordia.ca

Shin Hwei Tan†
Concordia University, Canada
shinhwei.tan@concordia.ca

***Abstract*—The rapid evolution of Web UI incurs time and effort in UI test maintenance. Prior techniques in Web UI test repair focus on locating the target elements on the new Webpage that match the old ones so that the corresponding broken statements can be repaired. These techniques usually rely on prioritizing certain attributes (e.g., XPath) during matching where the similarity of certain attributes is ranked before other attributes, indicating that there may be bias towards certain attributes during matching. To mitigate the bias, we present the first study that investigates the feasibility of using prior Web UI repair techniques for initial matching and then using ChatGPT to perform subsequent matching. Our key insight is that given a list of elements matched by prior techniques, ChatGPT can leverage language understanding to perform subsequent matching and use its code generation model for fixing the broken statements. To mitigate hallucination in ChatGPT, we design an explanation validator that checks if the provided explanation for the matching results is consistent, and provides hints to ChatGPT via a self-correction prompt to further improve its results. Our evaluation on a widely used dataset shows that the ChatGPT-enhanced techniques improve the effectiveness of existing Web test repair techniques. Our study also shares several important insights in improving future Web UI test repair techniques.**

***Index Terms*—Web UI Test Repair, Test Maintenance, UI Element Matching**

I. Introduction

When developers change a Web application's user interface (UI) for rapidly changing requirements, the corresponding Web UI tests need to be manually updated for test maintenance. To reduce manual efforts in repairing broken Web UI tests, academia and industry have proposed several approaches to automatically fix broken Web UI test cases caused by software evolution [1], [2], [3]. The key step in automated repair of Web UI tests is to modify broken statements containing outdated element locators by matching the element e_{old} in the old version of a Web application with the element e_{new} in the new version [4]. Prior Web UI test repair techniques mostly rely on a set of Document Object Model (DOM) attributes (e.g., identifiers and XPath) [1] or visual information [2] to determine whether the two elements e_{old} and e_{new} match. These techniques extract and compute the similarity of this information to select the most similar element as the result of the matching.

Due to numerous attributes available for matching Web UI elements, these techniques may prioritize certain attributes. For example, WATER [1], a classical Web UI test repair technique, applies a multi-step matching process using different attributes. First, it searches for identical elements by matching five attributes (*id*, *XPath*, *class*, *linkText*, *name*). If unsuccessful, it then finds similar DOM nodes using additional attributes. Specifically, it identifies elements with the same *tagname*, and computes their similarity scores between e_{old} and e_{new} with the weighted sum of *XPath* and other attributes where it prioritizes *XPath* similarity based on the heuristic that *XPath* of nodes "should be very similar across versions" [1]. As the prioritization and the predefined order on matching these attributes are usually based on heuristic of the tool developers, *the matching algorithm may not accurately reflect the evolution of the Web element*, causing mismatches, and subsequently failed repairs. Hence, it is important to understand the attribute prioritization used by prior test repair approaches. Meanwhile, prior learning-based techniques show promising results in combining different types of information for repairing broken Android GUI tests (e.g., combining word and layout embeddings [5], or fusing GUI structure and visual information [6]). The richer representation used by these learning-based techniques has been shown to enhance the accuracy of the UI matching step.

To solve the aforementioned problems of Web UI test repair and to hinge on a richer representation in learning-based approach, we present the *first study* of understanding the attribute prioritization and enhancing traditional Web UI test repair approaches with LLMs like ChatGPT. Our use of ChatGPT is motivated by the promising results shown in prior studies for solving related software maintenance tasks, (e.g., test generation [7], and automated program repair [8], [9]). However, the Web UI test repair problem differs from these tasks as it mainly involves Web element matching where accurate matching results typically lead to the correct repairs. Specifically, our study evaluates the effectiveness of integrating ChatGPT into two representative Web test repair techniques (WATER and VISTA [2]). To further evaluate the heuristic used in WATER that prioritizes XPath similarity, we also design a simplified variant of WATER that performs matching using only Levenshtein edit distance between XPaths of the old element e_{old} and the new element e_{new} (we call this approach EDIT DIS). Our study focuses on Java Selenium tests which are commonly used by Java Web applications.

The key insights of our approach are twofold: (1) we first rely on traditional Web UI test repair approaches to obtain

† Corresponding author.

an initial list of candidate matched elements (which may be biased by the prioritization used by the approach), and then use ChatGPT to perform subsequent matching to further select the best matched element in the candidate list, and (2) as ChatGPT may suffer from the hallucination problem [7], we design our prompt based on OpenAI's official documentation by asking ChatGPT to generate an explanation along with each selection. To mitigate hallucination, we propose an *explanation validator* that automatically checks for the consistency of the explanation, and instruct ChatGPT to self-correct the initial selection based on detected inconsistent explanation.

Our study aims to answer the following questions:

RQ1: Can ChatGPT help improve the accuracy of Web element matching of prior Web test repair approaches?

RQ2: What is the effectiveness of ChatGPT in repairing broken Selenium statements for Web test repair?

RQ3: When ChatGPT explains the result of the element matching, what is the quality of the explanation?

RQ4: Can our proposed explanation validator guide ChatGPT in self-correction to improve the matching and repair results?

Contributions. Our contributions are summarised as follows:

Study: We present the first study of understanding attribute prioritization and enhancing traditional test repair approaches with ChatGPT for Web UI test repair. Our findings include: (1) prior test repair tools have preferences towards certain attributes (e.g., XPath), (2) the combination with ChatGPT helps improve the matching accuracy of all evaluated approaches; (3) to our surprise, although EDIT DIS performs the worst individually, its combination with ChatGPT outperforms all the evaluated approaches in element matching and repair; (4) the repair effectiveness of all evaluated approaches is generally similar to the matching ones but there is one case where the repair performance decreases; (5) our proposed workflow of checking for explanation consistency and generating self-correct prompt as hint to ChatGPT could further improve the effectiveness for certain combinations (e.g., WATER+ChatGPT).

Technique: We proposed a novel Web UI test repair technique that uses traditional Web test repair approaches for initial matching and then uses ChatGPT for *subsequent matching* (uses all attributes of each UI element to perform another matching) to further improve the matching accuracy. Instead of prioritizing certain attributes in prior approaches (Table III), the subsequent matching steps use all attributes. To combat the hallucination problem in ChatGPT, we also design an explanation validator that automatically checks for the consistency of the generated explanation to guide self-correction.

Evaluation: We evaluate three baseline approaches (WATER, VISTA, EDIT DIS), their combinations with ChatGPT, and two recent approaches (WEBEVO and SFTM). Our evaluation on a widely used dataset [10] shows that the proposed workflow further improves the effectiveness of prior approaches in Web UI element matching and test repair.

II. BACKGROUND AND RELATED WORK

In this section, we introduce related work on test repair to provide a background on existing approaches.

Automatic UI test repair. Unlike traditional unit tests, UI tests are typically used to validate the functionality of particular UI application components through simulated user interactions such as button clicks. [1]. When a UI application evolves, the corresponding UI tests may crash, leading to significant effort required to manually fix these broken tests. Several techniques have been proposed for UI test maintenance [1], [2], [3], [11], [12], [13], [14], [5], [15], [16]. Most of these techniques focus on maintaining UI tests for mobile applications [11], [12], [6], [17], [5], Web applications [1], [2], [15], [3], [16] or desktop applications [13], [14]. Prior Web UI test repair techniques [1], [2], [3] typically (1) execute the test, (2) extract information from the Webpage based on various attributes [1] or visual information [2] of Web elements, (3) use various matching algorithms, and (4) update the locator of the matched element in the UI test to fix the broken test.

Model-based approaches [11], [12], [13], [14], [10] build a model of the application under test and modify the event flow to fix tests. Meanwhile, several heuristic-based approaches [1], [2], [3] relocate elements with UI matching algorithms and update broken tests via replacement of the matched elements. As heuristic-based approaches without building a model are more practical for large-scale applications, our paper mainly evaluates these approaches. A recent approach [5] first matches elements of two versions of Android apps using various similarity metrics (e.g., semantic embedding similarity, and layout similarity based on node embeddings of the GUI layout tree), and then repairs the tests by updating broken locators.

UI element matching. Given an element from an old version web, UI element matching aims to find the corresponding element in the new version. UI element matching has been applied to several domains (e.g., test reuse [18], [19], automated compatibility testing [20], and automated maintenance of UI tests). While several techniques exist for UI element matching, only a few are suitable for our evaluation. This paper evaluates classic UI test repair tools WATER [1] and VISTA [2], which match elements based on attribute information and visual information, respectively. Existing repair work mainly relies on strategies based on element information [1], [2], or simply combining machine learning algorithms with repair tools [5]. COLOR [15] and GUIDER [6] use a combination of attribute and visual information for matching. However, COLOR only returns updated locators instead of repairing broken tests while GUIDER focuses on Android test repair. Due to the differences between UI tests and existing APR work [21], we may not be able to directly use prior APR approaches to repair UI tests. Unlike prior approaches, we explore the feasibility of using prior test repair approaches to obtain an initial ranked list of elements and use ChatGPT to perform subsequent matching.

Instead of fixing broken Web UI tests, several approaches focus on improving the robustness of element locators. *Robula+* [22] focuses on generating robust XPath locators as XPath locators are sometimes the only option. *Robula+* checks whether an element or its ancestors possess unique attributes from a predefined whitelist (e.g., id, name, class). If multiple attributes are found, it chooses the attribute with the highest

priority to refine the XPath, thereby enhancing its robustness. Expanding on this, subsequent work [23] recommends multiple locator generators voting on the outcome, favoring reliable generators. Our design of the experiment is similar in essence with *Robula+*, letting ChatGPT make a selection among multiple candidates. However, our emphasis is on fixing broken locators instead of improving the robustness of the locators. While enhancing the robustness of locators is outside the scope of this paper, it could be worthwhile future work to explore LLM-based approaches for generating robust locators.

LLMs in software maintenance. LLMs such as ChatGPT have been applied for various software maintenance tasks [24], [25], [26], [7], [8], [9], including related tasks such as (1) test generation [7], [27], [28], [29], [30], [31], Android UI testing [32], [33], predicting flaky tests [34], and (2) automated program repair [8], [9], [24], [35]. ChatGPT is a transformer-based chatbot developed by OpenAI that helps developers create conversational AI applications [36]. Our proposed approach uses ChatGPT for (1) Web UI element matching, (2) providing explanations for the selected elements, and (3) repairing broken statements in tests. To improve code-related generation for LLM-based approaches, a recent approach was proposed using knowledge gained during the pre-training and fine-tuning stage to augment training data and their results show significant improvement for code summarization and code generation [26]. Similar to this approach, our proposed workflow also uses the experience gained during the selection of UI elements (i.e., inconsistency in the previous explanation generated by ChatGPT) to improve the UI matching results. Different from the aforementioned work, this paper focuses on using ChatGPT for solving the Web UI test repair problem that aims to perform UI test update by fixing broken UI element locators. To the best of our knowledge, our study also presents the first attempt that investigates and improves the quality of explanations provided by ChatGPT by designing an explanation validator to check the reliability of the explanation to improve the effectiveness of UI matching and test repair.

III. Motivating Examples

We show two examples where the combination of ChatGPT helps to improve the matching and test repair results.

Without self-correction. Table I shows a target element to be matched by WATER in the old version of the MRBS app, and key candidates from the returned list by WATER. Initially, WATER identifies elements with the same text as the target and returns the first match. However, in this case, two elements share the same value for the "text" attribute as the target element, and the second one has more similar XPath, position, and size to the target element than the first match, and ChatGPT correctly selected "Candidate 2" as the matching result from this ranked list. ChatGPT also explained that it chose this element "Because they share the most similar attributes: xpath, text, tagName, x, y, width, height." (its explanation consistency is 0.8, because all attributes except for XPath are consistent with the selection, and the most similar XPath is "/html[1]/body[1]" from another candidate).

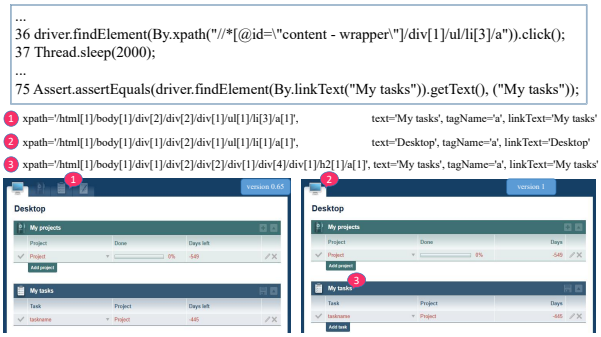

Fig. 1: Generated fixes for the broken statement in Collabtive that shows the effectiveness of self-correction.

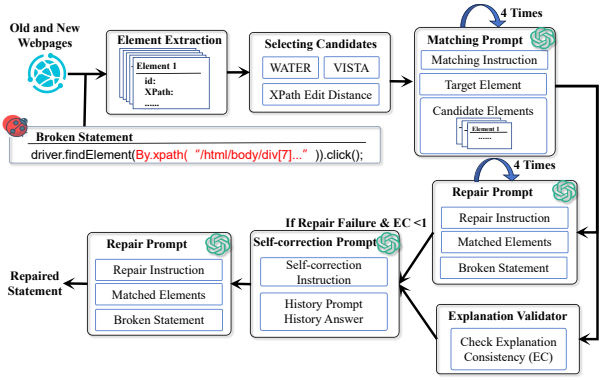

Fig. 2: Our proposed workflow of Web UI test repair

After matching, our approach fed ChatGPT with the repair prompt, and it successfully repaired the broken statement by updating the locator with XPath from "Candidate 2". In this example, with ChatGPT's explanation being 80% consistent, the subsequent matching step by ChatGPT helps select the correct but lower-ranked candidate, leading to successful repair.

With self-correction. Figure 1 shows the matched elements for another target element (①) selected by WATER+ChatGPT before and after self-correction. Initially, WATER+ChatGPT selected (②) with the explanation "Because they share the most similar attributes: xpath, text, tagName, linkText.". However, our explanation validator detects inconsistencies in the attributes *text* and *linkText* (Explanation Consistency=0.5) and provided this feedback via the self-correction prompt. Guided by this, WATER+ChatGPT revised its selection to (③) correctly, with a new explanation "Because they share the most similar attributes: xpath, text, tagName, linkText." (Explanation Consistency=0.75 as all attributes except XPath were consistent with the selection). This example highlights how the self-correction mechanism, supported by the explanation validator, guides ChatGPT to identify and fix its inconsistencies to improve matching results.

IV. Methodology

Figure 2 shows the workflow of our approach. First, our approach extracts information from the target element on the old Webpage based on the broken statement and all elements on the new Webpage. Then, it uses a test repair tool (WATER, VISTA or EDIT DIS) to rank the elements on the new Webpage, and

TABLE I: An example that shows the target element to be matched, and a list of candidate elements returned by WATER

Target Element	{numericId=96, id='', name='', class='', xpath='/html[1]/body[1]/p[1]', text='Unknown user', tagName='p', linkText='', x=8, y=90, width=1018, height=15, isLeaf=true}
Candidate 1	{numericId=40, id='', name='', class='', xpath='/html[1]/body[1]/div[1]/table[1]/tbody[1]/tr[1]/td[7]/div[1]/a[1]', text='Unknown user', tagName='a', linkText='Unknown user', x=917, y=6, width=111, height=23, isLeaf=true}
Candidate 2	{numericId=47, id='', name='', class='', xpath='**/html[1]/body[1]/div[2]/p[1]**', text='Unknown user', tagName='p', linkText='', **x=26, y=69, width=982, height=15**, isLeaf=true}
....	...

selects top-ranked ones for ChatGPT to perform further element matching. Our approach then instructs ChatGPT to select the matching element and explain its choice by mentioning the most similar attributes it considers. Based on the generated explanation, our explanation validator calculates the consistency of the explanation. As the accuracy of UI matching cannot be automatically validated without checking if the broken statement has been repaired correctly, our approach generates the repair prompt for ChatGPT to proceed with the repair by instructing ChatGPT to fix the broken statement based on its selected element. If the repair fails with explanation consistency (EC) less than 1, our approach generates a self-correctness prompt, our approach asks ChatGPT to provide another answer based on the inconsistencies in its original explanation. Finally, we obtain the repaired broken statement from ChatGPT. Due to the randomness of ChatGPT, we rerun the matching and repair process four times, and check if at least one correct repair occurs among the four runs, following the prior study [8].

A. Prompt Design of Repair and Self-Correction Prompt

We interact with ChatGPT via the API of the gpt-3.5-turbo model [37]. For optimal results, we design our prompt based on the official OpenAI documentations, including: (1) the official ChatGPT API Guide [38] and (2) OpenAI six strategies for achieving better results [39]. Table II shows for each sentence of the prompt (the "Prompt Content" column), the corresponding rule (the "Rule in OpenAI official documentation" column) that inspires the design. The ChatGPT API Guide emphasizes the importance of system instructions in providing high-level guidance for conversations so we use system instructions to design the Web UI test repair context part by telling ChatGPT that (1) it is a web UI test repair tool, and (2) outlining the steps for fixing a broken statement.

Due to token limits, we summarize ChatGPT's matched elements' information from previous dialogues to guide subsequent repair, aligning with the tactic "For dialogue applications that require very long conversations, summarize or filter previous dialogue." [39]. Following the rule "Use delimiters to clearly indicate distinct parts of the input" [39], we use colons to separate labels from respective contents, and curly braces to separate the information of each element, helping ChatGPT distinguish the input. Table II shows our prompt patterns and the applied rules. The complete prompt generation rules are:

Matching Prompt: Context[p1, p2, p3] + Input[p8]
Repair Prompt: Context[p1, p4, p5] + Input[p9]
Self-Correction Prompt: Context[p6, p7]

B. Information Extraction and Attribute Prioritization

To extract Web element information, we use an existing component in UITESTFIX [3] to extract relevant information for each element. Specifically, it retrieves the HTML source code of the target Webpage via a Web browser, then uses *Jsoup* library [41] to extract the Web elements' information in it. Note that we did not evaluate UITESTFIX's performance with other tools due to the unavailability of its matching algorithm. We extract information from several attributes: *id*, *name*, *class*, *XPath*, *text*, *tagName*, *linkText*, *x*, *y*, *width*, *height*, and *isLeaf*.

Understanding attribute prioritization. To gain a better understanding of the attribute prioritization used in prior techniques, we read the original paper and the implementation of each technique (if available). Table III shows the attributes and the priorities used by prior Web UI matching and repair approaches. The first two columns of Table III show the descriptions and examples of the attributes. The column "Used by Tools" shows if prior approaches WATER, VISTA, EDIT DIS, or UITESTFIX consider these attributes when matching elements and how these tools use them. We observe from Table III that *XPath* is the most commonly used attribute in all approaches. VISTA uses the fewest DOM attributes because it focuses on the image template matching algorithm. Two attributes (ie, *width* and *height*) are used to represent the size of the element. Meanwhile, WATER and UITESTFIX exclude these two attributes. WATER uses the greatest number of attributes, including *id*, *XPath*, *class*, *linkText*, *name*, *tagName*, *x* and *y*. UITESTFIX also uses four other attributes, of which *text* and *isLeaf* are not used by WATER. *isLeaf* is used by UITESTFIX toiteratively refining the similarity tthroughugh the DOM structure.

C. Candidate Selection

To address token limits, we provide ChatGPT with a list of 10 top-ranked candidate elements from the matching results of prior approaches (WATER,VISTA, EDIT DIS). We only chose 10 elements due to (1) the model's 4096-token limitation, and (2) the lengthy information of each element's 12 attributes. We briefly introduce the three approaches below:

VISTA: VISTA uses Fast Normalized Cross Correlation algorithm [42] to calculate the template matching results, defining the position and size of the matching area by the element's *x*, *y*, *width*, and *height*. It returns a list where elements of a particular screen position are ranked in descending order based on their similarity scores. Instead of returning only the top 1 element in the original version of VISTA, we modify VISTA to retrieve the top 10 elements as candidates for further matching.

TABLE II: The patterns of prompt and generation rules

PID	Rule in OpenAI's official documentation	Prompt Content
Web UI test repair context patterns: Context		
p1	Use system instruction to give high level instructions [38]	You are a web UI test script repair tool.
p2	Split complex tasks into simpler subtasks [40]	To repair the broken statement, you need to choose the element most similar to the target element from the given candidate element list. Give me your selected element's numericId and a brief explanation containing the attributes that are most similar to the target element.
p3	Provide examples [39]	Your answer should follow the format of this example: "The most similar element's numericId: 1. Because they share the most similar attributes: id, xpath, text."
p4	Summarize or filter previous dialogue [39]	To repair the broken statement, you chose the element *selected element* as the most similar to the target element from the given candidate element list.
p5	Specify the steps required to complete a task [39]	Now based on your selected element, update the locator and outdated assertion of the broken statement. Give the result of repaired statement.
p6	Use delimiters to clearly indicate distinct parts of the input [39]	This is a previous prompt: *Matching Prompt* This is your previous answer: *Corresponding Answer*
p7		But your explanation for attributes *attributes* are inconsistent with your selection and this will influence the correctness of your answer. Please answer again.
Input pattern: Input		
p8	Use delimiters to clearly indicate distinct parts of the input [39]	Target element: {*target element*} Candidate elements: {*candidate element 1*}, {*candidate element 2*} ...
p9		Broken statement: *broken statement*

TABLE III: Extracted attributes and the tools that use them. The superscript number indicates the priority of an attribute.

Attribute	Description (Example for an element)	Used by Tools
id	Unique identifier for an element (userName)	WATER^{E1} ,UITESTFIX^{E1}
name	Name for an element (submit)	WATER^{E5}
class	Class name for an element (button)	WATER^{E3}
XPath	Path of an element in the DOM tree (/html[1]/body[1]/div[1]/form[1]/ input[1])	WATER^{E2,L7}, EDIT DIS^{L1}, UITESTFIX^{T2}
text	Textual information of an element (Enter)	UITESTFIX^{L3}
tagName	Name of the HTML tag (input)	WATER^{E6}, UITESTFIX^{T2}
linkText	The text content of a hyperlink (Logout)	WATER^{E4}
x	X-coordinate for an element (66)	WATER^{8}
y	Y-coordinate for an element (183)	WATER^{8}
width	Width of an element (48)	VISTA^{P1}, WATER^{8}
height	Height of an element (21)	VISTA^{P1}, WATER^{8}
isLeaf	True if the element is a leaf node of the DOM tree, and false otherwise (true)	UITESTFIX^{4}

1 The superscript 'E' means the tool checks whether the candidate element's attribute is exactly the same as the target element.
2 The superscript 'L' indicates the tool uses Levenshtein Distance to calculate the similarity.
3 The superscript 'T' means that the tool uses TF-IDF to calculate the similarity.
4 The superscript 'P' means that the tool calculates the similarity of images, where the width and height of the images affect the calculation.
5 We refer to the **VISUAL** mode of VISTA [2] for VISTA's priority, **DOM** mode of VISTA [2] for WATER's priority.

WATER: As shown in Table III, WATER first checks if any candidate elements share the same *id*, *XPath*, *class*, *linkText*, or *name* as the target elements. and returns the first match. If none are found, WATER calculates similarity score for candidates with the same *tagName* as the target element, and returns the first one exceeding the threshold. The score combines the Levenshtein distance of their *XPath*, and the equivalence of their size and position based on *x*, *y*, *width*, and *height* with a small tolerance for variation. Notably, WATER gives a greater weight (0.9) to the *XPath* similarity because its authors assume that matched elements have similar *XPaths* after the version update. We modify WATER to return a de-duplicated list of 10 unique candidates instead of a single match.

EDIT DIS: Inspired by the idea of prioritizing XPath similarity in WATER, we design a simplified matching algorithm to only consider the XPath similarity. Similar to WATER, we use Levenshtein distance [43] to measure the difference between *XPaths*. Levenshtein distance measures the minimum number of insertions, deletions, and substitutions needed to transform one string into another. Greater distance means that the two elements are less similar. This algorithm has also been used in Web testing (e.g., detecting conflicting and outdated data on Webpages [44]). This variant returns the top 10 elements ranked in descending order based on their XPath similarities.

D. Explanation Validator

As shown in the "Prompt Content" column in Table II, we instruct ChatGPT to generate an explanation to describe the attributes used for selecting the best matched element. Our intuition is that *if the provided explanation is consistent, then the selection is more likely to be correct* (and repair is more likely to be successfully generated with correct matches). Based on this intuition, we designed an explanation validator to check if ChatGPT's explanation is consistent with the actual selection. Specifically, for each attribute a mentioned in the explanation, our explanation validator calculates the following to determine the most similar element for the consistency calculation $cons(a_i, R)$ where $cons(a_i, R)$=1 if the most similar element is selected:

Screen position: We use Euclidean distance to compute position-related attributes (e.g., x and y coordinates). We select the element with the minimum distance as the most similar.
Size: We use the product of width and height, and consider the one with the smallest size difference as the most similar.
isLeaf: We check if the *isLeaf* values (true or false) are same.
Other attributes: We use Levenshtein edit distance to measure similarity, with lower values indicating higher similarity.

In the cases where multiple candidate elements and the target element share the same similarity for a particular attribute, we will retain the results of multiple candidate elements and consider the explanation given by ChatGPT to be consistent if either one of these candidate elements has selected.

Definition 1 (Explanation Consistency (EC))**.** *Given the* target element t *(the element in the old version of the Webpage to be matched), the selection result* R *and an explanation* e *where* e *mentioned one or more attributes* $A = a_1, a_2, ..., a_n$*, we calculate the* Explanation Consistency (EC) *of* e *by computing cons(*a_i, R*) for each attribute* a_i *where* $cons(a_i, R)$ *checks whether each attribute* a_i *of* R *is most similar to that of the target element* t *(*$cons(a_i, R)$*=1 if the most similar element based on* a_i *is selected in* R*, and* $cons(a_i, R)$*=0 otherwise).*

$$EC(e) = \frac{\sum_{i=1}^{n} cons(a_i \in A, R)}{|A|}$$

Def. 1 presents the definition for Explanation Consistency (EC). For each explanation generated by ChatGPT, our explanation validator checks whether the attributes mentioned in the explanation that are similar between the selected element and the target element are consistent with the calculated values of all mentioned attributes. If the calculated values for all mentioned attributes are consistent, then our explanation validator considers the provided explanation as *consistent* across all mentioned attributes (EC=1).

V. Experimental Setup

Dataset. We use an existing dataset [10] to evaluate the effectiveness of Web test repair approaches. We use this dataset because (1) it is the only publicly available dataset for Web UI test repair, and (2) it has been widely used in prior evaluations of Web UI test repair approaches [3], [10]. The dataset contains Java Selenium UI tests from five open-source real-world Web applications (and VISTA [2] used 3 of them). All of these open-source applications are hosted in Sourceforge (except for MantisBT that is hosted in GitHub [45]). We follow the same filtering process of a prior evaluation [3] to remove duplicated tests and non-broken tests. Subsequently, we obtained 62 test cases containing 139 broken statements as our dataset. Each test script contains 1 to 6 broken statements, with an average of 70 lines of code per script.

Baselines selection. We evaluated three approaches (WATER, VISTA, EDIT DIS) that are widely used in prior studies of Web UI test repair. Similar to prior study [3], we exclude the model-based tool [10] as the provided code leads to compilation errors due to missing dependencies. We did not compare against several approaches [3], [16] as their matching algorithms are not publicly available so we could not evaluate their effectiveness in both matching and repair.

TABLE IV: Statistics of open-source Web apps in our dataset

Application	ΔV	Old Version	Updated Version	Tests	Broken Stmt
AddressBook	8	4.0	6.1	2	2
Claroline	29	1.10.7	1.11.5	27	53
Collabtive	5	0.65	1	4	11
MantisBT	38	1.1.8	1.2.0	25	66
MRBS	24	1.2.6.1	1.4.9	4	7
Avg/Total	21	-	-	62	139

Comparison with recent approaches. To assess the effectiveness of our approaches, we also evaluate two recent approaches: WEBEVO [46], which employs DOM tree-based change detection, history-based semantic structure change detection, and semantics-based visual search to match elements; and SFTM [47], which uses TD-IDF to calculate the initial similarity score on tokenized attributes and iteratively refines the matching based on the DOM structure. We did not combine these approaches with ChatGPT due to budget constraints.

We used the implementations of WATER, VISTA, WEBEVO, SFTM from UITESTFIX [3] for evaluation, which includes the re-implemented WATER by VISTA[2], and open-source implementations for the other tools.

Preparing ground truth dataset. As our evaluation dataset [10], [3] only has tests for the old versions of apps, and the test fixes are unavailable, we need to manually label the ground truths for the matching UI elements for the new versions. Specifically, two annotators independently labeled ground truths for each UI element located in the broken statement in the dataset for the three individual baselines (WATER,VISTA,EDIT DIS). The annotators are graduate students with over one year of experience in relevant research of Web UI test repair. For 12 cases out of 3*139=417 cases (139 for each of the 3 individual baselines), the annotators had disagreements and met to resolve.

Table IV shows the old and updated versions of the open-source Web apps in our dataset. The "ΔV" column denotes the number of versions between them where a greater number means more significant UI changes, posing more challenges to matching and repair tasks. The "Test" column denotes the number of tests for each app, whereas the "Broken Stmt" denotes the number of broken statements per app. Our experiment assumes each broken statement is independent so that we can measure the effectiveness of ChatGPT in fixing each broken statement. This assumption is similar to prior evaluations [48], [49] of learning-based automated repair techniques where the correct fault location is provided.

All experiments are run on a computer with an Intel Core i5 processor (1.6 GHz) and 12 GB RAM. For the experiments related to ChatGPT, we use the API of gpt-3.5-turbo model (which was the latest at the time of our experiments) and set the temperature to 0.8 as used in prior work [9].

VI. RQ1: Effectiveness of UI Matching

We evaluate the effectiveness of UI matching by calculating (1) the number of correct matches, and (2) ranking performance of the baselines.

TABLE V: The number of correct matching and repairs of different approaches. Bold values indicate the best performance.

Applications	Recent Approaches				Baselines and Combination Approaches (with self-correction)											
	WebEvo		SFTM		VISTA		VISTA+ChatGPT		WATER		WATER+ChatGPT		EDIT DIS		EDIT DIS+ChatGPT	
	Matching	Fix	Matching	Fix	Matching	Fix	Matching	Fix	Matching	Fix	Matching	Fix	Matching	Fix	Matching	Fix
AddressBook	0	0	**2**	**2**	1	1	1	1	**2**	**2**	**2**	**2**	0	0	**2**	**2**
Claroline	14	14	15	15	48	48	**51**	**51**	18	18	18	18	1	1	47	47
Collabtive	5	5	5	5	1	1	1	1	10	10	**11**	**11**	8	8	8	8
MantisBT	53	53	44	44	11	11	37	37	50	50	56	56	34	34	**64**	**63**
MRBS	2	2	5	5	**7**	**7**	**7**	**7**	1	1	4	4	0	0	2	2
Total	74	74	71	71	68	68	97	97	81	81	91	91	43	43	**123**	**122**

1) Matching Result and Analysis: We evaluate a total of eight matching approaches, including individual baselines (VISTA, WATER, EDIT DIS), their combinations with ChatGPT, and two recent approaches. For each approach, we recorded its abilities to correctly match the ground truth elements of the broken statements in our dataset. The "Matching" columns in Table V show the results on the evaluated applications. Comparing the overall matching results of individual baselines with their combination with ChatGPT (with self-correction), we observe that all combinations outperform the individual baseline (i.e., 97 versus 68 for VISTA, 91 versus 81 for WATER, and 123 versus 43 for EDIT DIS). This result confirms our hypothesis that combining prior test repair approaches with ChatGPT help improve the UI matching results. Across projects, the improvement is greatest in MantisBT with VISTA+ChatGPT and WATER+ChatGPT, in Claroline with EDIT DIS. Meanwhile, EDIT DIS+ChatGPT outperformed EDIT DIS in most projects, including AddressBook, Caroline, and MRBS. For the recent approaches, Table V shows that the WEBEVO and SFTM give similar matching results (74 and 71) that are slightly better than the two individual baselines (VISTA and EDIT DIS). However, both recent approaches perform worse than all the combination approaches, indicating that the combination with LLMs like ChatGPT can further enhance the matching accuracy of traditional approaches (WATER and VISTA).

Finding 1: All combinations of prior Web test repair approaches with ChatGPT performs generally better than the corresponding standalone approach (without ChatGPT).
Implication 1: Our suggested workflow of using prior repair tools for selecting candidate elements and then ChatGPT for subsequent matching help improve the matching accuracy.

We notice that VISTA's matching algorithm using visual information is effective for certain apps (e.g., Claroline and MRBS). However, EDIT DIS+ChatGPT performs the best among all approaches, fixing 122 out of a total of 139 broken statements. Specifically, EDIT DIS+ChatGPT yields the greatest improvement (186%) over the individual EDIT DIS approach. After combining with ChatGPT, it correctly matches 80 elements that were originally mismatched and ensures that nearly all initially correctly matched statements remain correct (42). Given that the individual EDIT DIS performs the worst among all baselines, we think this result is counterintuitive as one would select the tool that performs well individually (i.e., WATER) to combine with ChatGPT to get more improvement.

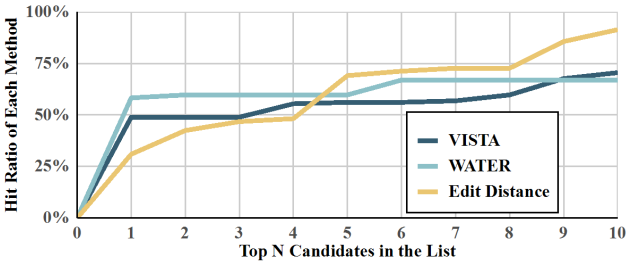

Fig. 3: Comparison of Hit Ratios for the three baselines

Finding 2: Although the individual EDIT DIS approach performs the worst among all individual baselines, its combination with ChatGPT outperforms all evaluated approaches.
Implication 2: EDIT DIS combines well with ChatGPT. By prioritizing only XPath similarity, it is more effective in guiding ChatGPT for subsequent matching.

Effectiveness of the standalone ChatGPT (ChatGPT only). Another natural baseline approach will be to use the standalone ChatGPT for performing all the matching and repair steps. Hence, we also check the effectiveness of standalone ChatGPT by providing all UI elements as candidates instead of using a selection algorithm to choose 10 of them. However, all the prompts throw errors due to the token limit being exceeded for all cases. This is expected as for a Webpage in our dataset, there are an average of 224 UI elements to be matched where attribute information for each element occupies around 101 tokens. The average number of tokens (224*101=21733) also shows the high cost of the standalone ChatGPT.

2) Ranking performance of the baselines: As the combination of EDIT DIS+ChatGPT outperforms all other approaches, we investigate the reasons behind the improvement. Specifically, as WATER and VISTA originally return only one element as the best matching result, we analyze the ranking performance of each baseline approach. Given a selected element s_e by a tool t and the correct element t_e (i.e., the corresponding element in the ground truth), we consider t *hits* if s_e is exactly the same as t_e. If one of the elements in ranked list produced by a tool t hits, we record its ranking to evaluate the ranking performance. We employ metrics commonly used in evaluating top-N recommendation task [50]: Top-K Hit Ratio (HR, or Recall, in this study, the proportion of experimental instances in which the top N candidates selected by each baseline contain ground truth in the candidate list). Figure 3 depicts the variation

of Hit Ratio with increasing values of N for the three baselines. We observe several interesting trends in Figure 3: 1) When N=1, the Hit Ratio for VISTA and WATER is already around 50%, while EDIT DIS's is only 30.9%. This indicates that the top candidate of VISTA and WATER already effectively hits ground truth, whereas EDIT DIS's advantage is less evident. 2) As N increases, the Hit Ratios for VISTA and WATER increase slowly. Even at N=10, their Hit Ratios only increase by less than 25% compared to when N is 1. In contrast, EDIT DIS's Hit Ratio gradually increases, reaching 68.3%, 84.2%, and 90.1% when N is 5, 9, and 10, respectively. Hence, if expanding the number of selected candidates to the top 10 in the candidate list, EDIT DIS is more likely to hit the ground truth element compared to VISTA and WATER, leading to more potential improvement when combined with ChatGPT.

VII. RQ2: EFFECTIVENESS AND EFFICIENCY OF REPAIR

Before checking for the repair correctness of each repaired statement, we first check if ChatGPT has the correct matching result for the broken statement because a correct repair can only be generated after a correct matching.

Repair Correctness. As the ground truth repaired statements written by developers of the Web apps are unavailable in our dataset, and the repaired statements generated by ChatGPT may have diverse but semantically-equivalent fixes, we need to manually validate the correctness of all generated repaired statements. To reduce the manual effort in validation, we use a semi-automated approach. Specifically, given the original broken statement o, the repaired statement r, and the ground truth element e, we design a parser that automatically parses the locator type (e.g., `By.name` and `By.xpath`), and the expression within the locator (e.g., the XPath value) in the repaired statement r to verify their correctness with respect to the ground truth element e. For example, if ChatGPT uses `By.xpath` as the locator, it should use the XPath information of e rather than the value of other attributes. Our parser also identifies cases requiring manual analysis where more substantial changes have been introduced by ChatGPT, including (1) different types of locators between o and r, and (2) additional statements added to the repaired statement r.

The "Fix" columns in Table V show the repair results for all approaches. The combination approaches (with self-correction) yield the best results across four runs. On average, VISTA+ChatGPT, WATER+ChatGPT, and EDIT DIS+ChatGPT achieve success in 2.17, 2.01, and 2.89 runs out of 4, respectively, indicating consistent performance across different runs. From Table V, we can observe that the number of correct repair is almost equal to the number of correct matching, except for one case in EDIT DIS+ChatGPT. For MantisBT, EDIT DIS+ChatGPT correctly matches an element but generates a repair that modifies the original intention of the broken statement. Specifically, the broken statement sends an empty text input "" to the element, but EDIT DIS's repaired statement sends "Test" to the element. Nevertheless, EDIT DIS+ChatGPT still excels with the best performance (122 correct repairs).

Repair Efficiency. To assess potential delays from using ChatGPT, we measured the total time for matching and repair for each broken statement. On average, WEBEVO takes 45.73 s, SFTM takes 47.98 s, VISTA takes 36.91 s, VISTA+ChatGPT 41.44 s, WATER takes 30.22 s, WATER+ChatGPT 35.11 s, EDIT DIS 24.02 s, and EDIT DIS+ChatGPT 28.17 s per broken statement. Approaches involving VISTA and WEBEVO are more time-consuming because VISTA uses computer vision techniques for matching, which takes more time than matching textual information. SFTM takes more time because it iteratively propagates and refines the similarity scores during matching process. Overall, ChatGPT adds minimal overhead.

Finding 3: The repair effectiveness is mostly similar to the matching effectiveness (except for one incorrect repair).
Implication 3: Using correctly matched elements, most approaches could repair correctly.

VIII. RQ3: QUALITY OF CHATGPT'S EXPLANATION

As our study uses explanation to overcome hallucination, it is important to investigate its quality. To assess the quality of ChatGPT's explanations for element matching results, we use two metrics: (1) *mention frequency* (the number of times where an attribute a has been mentioned M), and (2) *mention consistency* (the number of times where an attribute a has been consistently mentioned C where the consistency is determined by our explanation validator described in Section IV-D). These two metrics helps in answering the research questions below:

RQ3a: What are the frequently mentioned attributes in ChatGPT's explanation?
RQ3b: What are the mentioned attributes that are consistent in ChatGPT's explanation?

Table VI presents the mention frequency and mention consistency of ChatGPT for each attribute. Table VI shows that ChatGPT mentions two attributes most frequently: *XPath* (1660) and *text* (1462), whereas the least mentioned attribute is *id* (88). Compared to the priority imposed by prior test repair approaches (shown in Table III), this result shows that ChatGPT has certain preferences towards particular attributes since it often mention the *XPath* and *text* attributes regardless of the baseline used for selecting the list of candidate elements.

Finding 4: ChatGPT frequently mentions the the *XPath* and *text* attributes in the provided explanations.
Implication 4: Similar to other approaches that have certain priority, ChatGPT prioritizes *XPath* and *text* attributes.

Table VI also shows the attributes with the greatest mention consistency are *text* (1026), *XPath* (995), and *tagName* (995). Although ChatGPT prefers using *XPath* (1660) and *text* (1462) for matching, our results show that prioritizing the *text* attribute over the *XPath* attribute will lead to better matching results (as the *text* attribute is mentioned more consistently).

TABLE VI: Effectiveness of SC Mechanism: "M" represents the number of times an attribute is mentioned and "C" denotes the number of times where the mentioned attribute is consistent in the generated explanation for each approach

Approach	id		name		class		XPath		text		tagName		linkText		position		size		isLeaf	
	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M
VISTA+ChatGPT	10	35	21	50	112	146	393	590	302	497	368	424	115	135	85	217	131	218	175	188
WATER+ChatGPT	14	25	40	75	116	152	276	547	362	498	360	367	180	203	157	241	226	241	212	215
EDIT DIS+ChatGPT	25	28	24	29	123	144	326	523	362	467	267	309	126	160	137	251	186	249	191	197
Total	49	88	85	154	351	442	995	**1660**	**1026**	1462	995	1100	421	498	379	709	543	708	578	600

TABLE VII: The results before and after self-correction (SC)

Approach	VISTA+ChatGPT		WATER+ChatGPT		EDIT DIS+ChatGPT	
	before SC	after SC	before SC	after SC	before SC	after SC
Matching	97	97	86	91	122	123
Repair	97	97	86	91	121	122

Finding 5: The most frequently mentioned attribute by ChatGPT (i.e., XPath) tends to lead to incorrect matching (low mentioned consistency). In contrast, the *text* attribute is high in mention frequency and mention consistency.
Implication 5: Prioritizing the *text* attribute over *XPath* attribute is better for ChatGPT as it will be more accurate.

Correlation between EC and correctness of the matching. Our explanation validator measures the explanation consistency (EC) as a mechanism to check and improve the reliability of ChatGPT's matching results. However, even if our explanation validator can accurately assess EC, it does not guarantee correct matching (i.e., retrieving the target element in the labeled ground truth). To investigate the correlation between our proposed EC and the *correctness of the final matching result* (i.e., whether it selects the ground truth element), we measure the correlation between these two variables. As the two variables are binary (correctness) and continuous (EC between 0 and 1) categories respectively, we compute the Point-Biserial Correlation Coefficient (r_{pbi}) [51] for all explanations. As this is a special case of the Pearson Correlation, we assess the strength of the relationship by calculating the correlation coefficient. For the three combinations with ChatGPT, the values for the Point-Biserial Correlation Coefficient are: r_{pbi}=0.51 for VISTA+ChatGPT, r_{pbi}=0.84 for WATER+ChatGPT, and r_{pbi}=0.49 for EDIT DIS+ChatGPT. These values indicate that EC and the matching correctness for VISTA+ChatGPT and EDIT DIS+ChatGPT are only moderately correlated but *strongly correlated for the* WATER+*ChatGPT combination*.

Finding 6: EC for WATER+ChatGPT strongly correlates with matching correctness among all combinations.
Implication 6: The strong correlation shows that our explanation validator is the most effective for WATER+ChatGPT to improve its matching correctness.

IX. RQ4: Ablation study for the SC mechanism

As the self-correction (SC) mechanism was the key in the design of our proposed combination, we conduct an ablation study in RQ4 to evaluate the effectiveness of this mechanism. Specifically, we count and compare the number of correct matches and repairs before and after the self-correction. Table VII presents the results of the three approaches combined with ChatGPT before and after self-correction (SC), comparing the correct matches and average explanation consistency (EC). Except for EDIT DIS+ChatGPT, the number of correct matches has increased for the other approaches. Notably, WATER+ChatGPT shows the most significant improvement, gaining 5 more correct matches and repairs after self-correction (SC). Meanwhile, we think that EDIT DIS+ChatGPT does not show any improvement due to the moderately positive correlation between EC and matching accuracy (0.49).

Finding 7: The improvement given by self-correction varies across tools. Among the three combinations, WATER benefits the most (5 more correct matching after correction).
Implication 7: Our proposed workflow of guiding ChatGPT via self-correct prompt improves the effectiveness of certain combinations (e.g., WATER+ChatGPT).

X. Implications and Discussions

Our study identifies several key implications and suggestions for future test repair and ChatGPT research.

Prioritization of attributes by Web UI test repair tools. Table III shows varying attribute prioritization in element matching among prior test repair tools (e.g., WATER prioritizes XPath similarity, whereas VISTA uses the position and the size information for visual matching). Our study shows that further matching with ChatGPT can improve over prior approaches (Finding 1) by mitigating their bias, leading to more accurate matching. In fact, our study of the frequently mentioned attributes in ChatGPT's explanation also reveals that ChatGPT has preferences towards certain attributes, e.g., *XPath* and *text* (Finding 5). Although this paper only studies the prioritization of attributes in two traditional Web test repair approaches, similar biases may exist in other UI matching techniques. In the future, it is worthwhile to study (1) the characteristics of the selected prioritization in other tasks where UI matching algorithms are used (e.g., compatibility testing [20]), and (2) improving the effectiveness of other UI matching techniques via subsequent matching using LLMs.

Test repair techniques used for element pre-selection. Our study that compares three approaches (WATER,VISTA, and EDIT DIS) for the pre-selection of candidate elements shows that a simplified version of WATER (i.e., EDIT DIS) combines well with ChatGPT where EDIT DIS+ChatGPT outperforms all

the evaluated approaches (Finding 2). Compared to WATER that matches multiple attributes (e.g., *id*, *XPath*) and VISTA that uses visual information for matching, EDIT DIS that solely relies on *XPath* is less effective as a standalone matching algorithm (it performs worst among all individual baselines). However, our study shows that by using only XPath similarity, EDIT DIS delegates the responsibility of matching using other attributes to ChatGPT which later performs subsequent matching. Intuitively, one may think that WATER that performs the best among the individual baselines would lead to the best results when combining with ChatGPT. *As the best individual baseline may not be the most effective combination with ChatGPT*, our study urges researchers to perform thorough evaluation to choose appropriate baseline to combine with ChatGPT for solving other software maintenance tasks.

LLM-based test repair and robust locators generation. Our study shows promising results in using LLMs like ChatGPT for Web test repair (Finding 3). Our manual analysis shows that it can generate correct repairs for the broken statements. Currently, the generated fixes mainly modifies and generate assertions, showing the promises of LLM-based test repair. In future, it is worthwhile to study using LLM for improving test repair technique that fixes broken assertions [52] [53]. Another worthwhile future work is to use LLMs like ChatGPT to improve the locator robustness and incorporated into prior techniques like *Robula+* [22]. For robust locators generation, ChatGPT may be instructed to select different locators, prioritizing locators that are less fragile.

Improving reliability of ChatGPT output. Prior study has expressed concerns regarding the tendency of ChatGPT to "hallucinate" when solving specific tasks [7], our workflow (that checks whether the explanations provided by ChatGPT are consistent along with the selected elements) shows promising results for improving the matching accuracy for certain combinations (Finding 7). Although our study are limited to Web UI test maintenance, we believe that our proposed way of checking for explanation consistencies is general and can be applied to improve the reliability of ChatGPT for other software maintenance tasks (e.g., test generation).

XI. THREATS TO VALIDITY

External Threats. During the ground truth construction and evaluating our way of calculating EC, we mitigate potential bias by asking two annotators to manually construct and cross-validate the "ground truth target element and patch". The two annotators met to resolve any disagreement during the annotation, and further discussed until a consensus was reached. To reduce bias in selection of Web applications, we evaluate on a widely used dataset. As with prior evaluations, our findings may not generalize beyond the selected applications and tests in the dataset. To encourage future research in Web UI element matching and repair, we also release our dataset. Due to limited resources and budget, we use the cost-effective GPT 3.5-Turbo model which may be less effective than newer models in Web test repair. Nevertheless, the extended input of a ChatGPT with a larger token limit (e.g., ChatGPT-16k) may still be insufficient for representing UI elements because for a Webpage, there are an average of 224 elements to be matched where the attribute information for each element occupies 101 tokens, 224*101=22624 tokens. As ChatGPT performance may vary across settings, our experiments may not generalize beyond the studied settings, and fixing other forms of UI tests (we focus on Java Selenium Web UI tests [54]). We mitigate this threat by reusing settings and suggestions in prior work (e.g., referring to OpenAI documentation for prompt design and using temperature 0.8 as in prior work [9]), and evaluating several test repair tools that use different algorithms (e.g., text-based and visual-based). As our study only evaluates the test repair capability of ChatGPT, the findings may not apply for other LLMs. Nevertheless, our suggested workflow of using LLMs to perform subsequent matching and rematching based on inconsistent explanations are still generally applicable.

Internal Threats. Our experimental scripts may have bugs that can affect our results. To mitigate this threat, we made our scripts and results publicly available [55].

Conclusion Threats. Conclusion threats of our study include (1) overfitting of our dataset and (2) subjectivity of ground truth construction. To ensure that the matching and repair tasks do not overlap with the training dataset of ChatGPT to minimize the possibility of overfitting, we manually analyze the updated UI tests, and we have manually labeled and created a ground truth dataset that can be used to support future research in Web UI test repair. We mitigate the subjectivity of ground truth construction by cross-validating between two annotators during ground truth construction.

XII. CONCLUSIONS

This paper presents the first feasibility study that evaluates the effectiveness of using prior Web UI repair techniques for initial matching and then using ChatGPT to perform subsequent matching to mitigate the bias in prioritization of attributes of prior approaches. To reduce hallucination in ChatGPT, we introduce an explanation validator that checks the consistency of the provided explanation, and gives hints to ChatGPT via a self-correction prompt to further improve its results. Our evaluation on a widely used dataset shows that the combinations with ChatGPT improve the effectiveness of prior techniques. Our study also reveals several findings and implications. As an initial study that focuses on LLM-based Web test repair, we hope that our study could shed light in improving future Web UI test repair approaches.

XIII. ACKNOWLEDGEMENTS

We greatly appreciate the effort of Yuanzhang Lin for his help in annotating the ground truth fixes. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants (RGPIN-2024-04301).

DATA AVAILABILITY

To ensure reproducibility and advance future research in test repair, we made the dataset, scripts and experimental data publicly available on Github under an MIT license [55].

REFERENCES

[1] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, 2011, pp. 24–29. [Online]. Available: https://doi.org/10.1145/2002931.2002935

[2] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 503–514. [Online]. Available: https://doi.org/10.1145/3236024.3236063

[3] Y. Lin, G. Wen, and X. Gao, "Automated fixing of web UI tests via iterative element matching," in *38th IEEE/ACM International Conference on Automated Software Engineering*, 2023. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00048

[4] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of web applications break?" in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 180–190. [Online]. Available: https://doi.org/10.1109/ICST.2016.16

[5] J. Yoon, S. Chung, K. Shin, J. Kim, S. Hong, and S. Yoo, "Repairing fragile gui test cases using word and layout embedding," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 291–301. [Online]. Available: https://doi.org/10.1109/ICST53961.2022.00038

[6] T. Xu, M. Pan, Y. Pei, G. Li, X. Zeng, T. Zhang, Y. Deng, and X. Li, "Guider: Gui structure and vision co-guided test script repair for android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 191–203. [Online]. Available: https://doi.org/10.1145/3460319.3464830

[7] R. Feldt, S. Kang, J. Yoon, and S. Yoo, "Socratest: Towards autonomous testing agents via conversational large language models," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00148

[8] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2023, pp. 23–30. [Online]. Available: https://doi.org/10.1109/APR59189.2023.00012

[9] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128

[10] J. Imtiaz, M. Z. Iqbal, and et al., "An automated model-based approach to repair test suites of evolving web applications," *Journal of Systems and Software*, vol. 171, p. 110841, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110841

[11] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "Atom: Automatic maintenance of gui test scripts for evolving mobile applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 161–171. [Online]. Available: https://doi.org/10.1109/ICST.2017.22

[12] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, "Change-based test script maintenance for android apps," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 215–225. [Online]. Available: https://doi.org/10.1109/QRS.2018.00035

[13] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "Sitar: Gui test script repair," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 170–186, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2015.2454510

[14] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving gui applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 45–55. [Online]. Available: https://doi.org/10.1145/2483760.2483775

[15] H. Kirinuki, H. Tanno, and K. Natsukawa, "Color: correct locator recommender for broken test scripts using various clues in web application," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 310–320. [Online]. Available: https://doi.org/10.1109/SANER.2019.8667976

[16] W. Chen, H. Cao, and X. Blanc, "An improving approach for dom-based web test suite repair," in *Web Engineering*. Springer International Publishing, 2021, pp. 372–387. [Online]. Available: https://doi.org/10.1007/978-3-030-74296-6_29_WTSR

[17] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "Gui-guided test script repair for mobile apps," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2020.3007664

[18] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching of GUI events for test reuse: Are we there yet?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: Association for Computing Machinery, 2021, p. 177–190. [Online]. Available: https://doi.org/10.1145/3460319.3464827

[19] F. Khalili, L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching in GUI test reuse," *Empirical Software Engineering*, vol. 29, no. 3, pp. 1–58, 2024. [Online]. Available: https://doi.org/10.1007/s10664-023-10406-8

[20] Y. Ren, Y. Gu, Z. Ma, H. Zhu, and F. Yin, "Cross-device difference detector for mobile application gui compatibility testing," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2022, pp. 253–260. [Online]. Available: https://doi.org/10.1109/ICSTW55395.2022.00052

[21] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[22] F. R. M. Leotta, A. Stocco and P. Tonella, "Robula+: An algorithm for generating robust xpath locators for web testing," in *J. Softw. Evol. Process*, vol. 28, no. 3, 2016, pp. 177–204. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[23] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Using multi-locators to increase the robustness of web test cases," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICST.2015.7102611

[24] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509. [Online]. Available: https://doi.org/10.1109/MSR52588.2021.00063

[25] T. Xiao, C. Treude, H. Hata, and K. Matsumoto, "Devgpt: Studying developer-chatgpt conversations," in *Proceedings of the International Conference on Mining Software Repositories (MSR 2024)*, 2024. [Online]. Available: https://doi.org/10.1145/3643991.3648400

[26] H. Q. To, N. D. Bui, J. Guo, and T. N. Nguyen, "Better language models of code through self-improvement," *arXiv preprint arXiv:2304.01228*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.01228

[27] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 919–931. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00085

[28] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "LLM for test script generation and migration: Challenges, capabilities, and opportunities," *arXiv preprint arXiv:2309.13574*, 2023. [Online]. Available: https://doi.org/10.1109/QRS60937.2023.00029

[29] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative ai," *arXiv preprint arXiv:2306.06782*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.06782

[30] K. El Haji, C. Brandt, and A. Zaidman, "Using github copilot for test generation in python: An empirical study," in *The 5th ACM/IEEE International Conference on Automation of Software Test*, 2024. [Online]. Available: https://doi.org/10.1145/3644032.3644443

[31] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer, "Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 14–26. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00089

[32] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1355–1367. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00119

[33] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser.

ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608137

[34] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1912–1927, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3201209

[35] H. Joshi, J. Cambronero Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with llms," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, pp. 5131–5140, Jun. 2023. [Online]. Available: https://doi.org/10.1609/aaai.v37i4.25642

[36] R. Santhosh, M. Abinaya, V. Anusuya, and D. Gowthami, "Chatgpt: Opportunities, features and future prospects," in *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI)*, 2023, pp. 1614–1622. [Online]. Available: https://doi.org/10.1109/ICOEI56765.2023.10125747

[37] OpenAI, "Openai gpt-3 api documentation," https://platform.openai.com/docs/api-reference, 2022.

[38] OpenAI Help. (2023) Chatgpt api transition guide. Accessed: November 15, 2023. [Online]. Available: https://help.openai.com/en/articles/7042661-chatgpt-api-transition-guide

[39] OpenAI. (2023) Six strategies for getting better results with gpt. Accessed: November 15, 2023. [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results

[40] OpenAI. (2023) Strategy: Split complex tasks into simpler subtasks. Accessed: November 15, 2023. [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering/strategy-split-complex-tasks-into-simpler-subtasks

[41] P. Houston, *Instant Jsoup How-To*, ser. Instant : short, fast, focused. Packt Publishing, 2013. [Online]. Available: https://books.google.ca/books?id=24I9oaxuvowC

[42] K. Briechle and U. D. Hanebeck, "Template matching using fast normalized cross correlation," in *Optical Pattern Recognition XII*, vol. 4387. SPIE, 2001, pp. 95–102. [Online]. Available: https://doi.org/10.1117/12.421129

[43] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, 1998. [Online]. Available: https://doi.org/10.1109/34.682181

[44] N. Jnoub, W. Klas, P. Kalchgruber, and E. Momeni, "A flexible algorithmic approach for identifying conflicting/deviating data on the web," in *2018 International Conference on Computer, Information and Telecommunication Systems (CITS)*, 2018, pp. 1–5. [Online]. Available: https://doi.org/10.1109/CITS.2018.8440185

[45] "Mantisbt bug tracker repository," https://github.com/mantisbt/mantisbt, 2022.

[46] F. Shao, R. Xu, W. Haque, J. Xu, Y. Zhang, W. Yang, Y. Ye, and X. Xiao, "Webevo: taming web application evolution via detecting semantic structure changes," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 16–28. [Online]. Available: https://doi.org/10.1145/3460319.3464800

[47] S. Brisset, R. Rouvoy, L. Seinturier, and R. Pawlak, "Sftm: Fast matching of web pages using similarity-based flexible tree matching," *Information Systems*, vol. 112, p. 102126, 2023. [Online]. Available: https://doi.org/10.1016/j.is.2022.102126

[48] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 511–523. [Online]. Available: https://doi.org/10.1145/3510003.3510177

[49] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[50] D. Li, R. Jin, J. Gao, and Z. Liu, "On sampling top-k recommendation evaluation," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2114–2124. [Online]. Available: https://doi.org/10.1145/3394486.3403262

[51] R. F. Tate, "Correlation between a discrete and a continuous variable. point-biserial correlation," *The Annals of Mathematical Statistics*, vol. 25, no. 3, pp. 603–607, 1954. [Online]. Available: http://www.jstor.org/stable/2236844

[52] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov, "Reassert: a tool for repairing broken unit tests," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1010–1012. [Online]. Available: https://doi.org/10.1145/1985793.1985978

[53] A. S. Yaraghi, D. Holden, N. Kahani, and L. Briand, "Automated test case repair using language models," *arXiv preprint arXiv:2401.06765*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.06765

[54] S. M. Shariff, H. Li, C.-P. Bezemer, A. E. Hassan, T. H. Nguyen, and P. Flora, "Improving the testing efficiency of selenium-based load tests," in *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. IEEE, 2019, pp. 14–20. [Online]. Available: https://doi.org/10.1109/AST.2019.00008

[55] Z. Xu, "Chatgpt ui test repair repository," https://github.com/xuzhi2021/ChatGPTUITestRepair, 2025.